Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science 142 (2006) 31–46
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Foundations for Designing Secure Architectures

Jan Jürjens[1]

Competence Center for IT Security
Software & Systems Engineering, TU München, Germany

Abstract

Developing security-critical systems is difficult and there are many well-known examples of security weaknesses exploited in practice. In particular, so far little research has been performed on the soundly based design of secure architectures, which would be urgently needed to develop secure systems reliably and efficiently. In this abstract, we sketch some research on a sound methodology supporting secure architecture design. We give an overview over an extension of UML, called UMLsec, that allows expressing security-relevant information within the diagrams in an architectural design specification. We define foundations for secure architectural design patterns. We present tool-support which has been developed for the UMLsec secure architecture approach.

Keywords: Secure software engineering, secure architectures, security engineering, security verification, formal methods in security, security evaluation, security models, cryptographic protocols

1 Motivation

The high quality development of security-critical systems is difficult. Many critical systems are developed, deployed, and used that do not satisfy their security requirements, sometimes allowing spectacular attacks. In particular, so far little research has been performed on the soundly based design of secure architectures, which would be urgently needed to develop secure systems

---

[1] http://www4.in.tum.de/~juerjens . This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author(s).

doi:10.1016/j.entcs.2005.07.012

reliably and efficiently. Part of the difficulty of secure systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

Within the field of Software Architectures [2], the Unified Modeling Language (UML) has been proposed to be used also as an Architecture Description Language (ADL). In particular, UML offers an unprecedented opportunity for high-quality secure systems development that is feasible in an industrial context.

- As the de-facto standard in industrial modeling, a large number of developers is trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.
- A number of analysis, testing, simulation, transformation and other tools are developed to assist the every-day work using UML.

This article, which is based on the tutorial [6] on the same topic, gives a short introduction into using the formally based UML security extension UMLsec to develop foundations for designing secure architectures. Firstly, we recall the definition of a simplified fragment of UMLsec and its formal foundation to be used in this paper. We sketch how one can use stereotypes, tags, and constraints to encapsulate knowledge on secure architectures and thereby make it available to developers which may not be specialized in secure systems. In particular, we explain how to use the formally based UMLsec to provide foundations for secure architectural design patterns. We also demonstrate how one can formally verify whether the constraints associated with the stereotypes are fulfilled in a given secure architecture design using the tool-support provided. This way one can find flaws present in the design before a system is deployed, or even implemented. Finally, we explain how to provide foundations for designing secure systems based on the Java Security Architecture.

2 Formal Basis for Secure Architecture Analysis

We shortly recall the formal basis of UMLsec from [5] which will be used for secure architecture analysis in the later sections. More details can be found in [5].

Outline of Formal Semantics

For some of the constraints used to define the UMLsec extension we need to refer to a precisely defined semantics of behavioral aspects. For security analysis, the security-relevant information from the security-oriented stereotypes is then incorporated.

Our formal semantics of a simplified fragment of UML includes activity diagrams, statecharts, sequence diagrams, static structure diagrams, deployment diagrams, and subsystems, simplified to keep a formal treatment that is necessary for some of the more subtle security requirements feasible. The subsystems integrate the information between the different kinds of diagrams and between different parts of the system specification. We only outline the basic concepts, a complete account is in [5], which also includes pointers to earlier work on which this work is based.

In UML the objects or components communicate through messages received in their input queues and released to their output queues. Thus for each component C of a given system, our semantics defines a function $[\![C]\!]()$ which

- takes a multi-set I of input messages and a component state S and
- outputs a set $[\![C]\!](I, S)$ of pairs (O, T) where O is a multi-set of output messages and T the new component state (it is a *set* of pairs because of the non-determinism that may arise)

together with an *initial state* S_0 of the component.

The behavioral semantics $[\![D]\!]()$ of a statechart diagram D models the run-to-completion semantics of UML statecharts. As a special case, this gives us the semantics for activity diagrams. Given a sequence diagram $\mathcal{S}$, we define the behavior $[\![\mathcal{S}.C]\!]()$ of each contained component C.

Subsystems group together diagrams describing different parts of a system: a system component $\mathcal{C}$ given by a subsystem $\mathcal{S}$ may contain subcomponents $\mathcal{C}_1, \ldots,$
$\mathcal{C}_n$. The behavioral interpretation $[\![\mathcal{S}]\!]()$ of $\mathcal{S}$ is defined as follows:

(1) It takes a multi-set of input events.

(2) The events are distributed from the input multi-set and the link queues connecting the subcomponents and given as arguments to the functions defining the behavior of the intended recipients in $\mathcal{S}$.

(3) The output messages from these functions are distributed to the link queues of the links connecting the sender of a message to the receiver, or given as the output from $[\![\mathcal{S}]\!]()$ when the receiver is not part of $\mathcal{S}$.

When performing security analysis, after the last step, the adversary model may modify the contents of the link queues in a certain way explained below.

Security Analysis

For a security analysis of a given UMLsec subsystem specification $\mathcal{S}$, we need to model potential adversary behavior. We model specific types of adversaries that can attack different parts of the system in a specified way. For this we assume a function $\mathsf{Threats}_A(s)$ which takes an *adversary type* A and a stereotype s and returns a subset of $\{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$. Then we model the actual behavior of an adversary of type A as a *type A adversary function* that non-deterministically maps the contents of the link queues in $\mathcal{S}$ and a state S to the new contents of the link queues in $\mathcal{S}$ and a new state T:

- the contents of links stereotyped s where $\mathsf{delete} \in \mathsf{Threats}_A(s)$ may be mapped to $\emptyset$ and
- the contents of links stereotyped s where $\mathsf{insert} \in \mathsf{Threats}_A(s)$ may be enlarged by elements from the contents of links stereotyped t where $\mathsf{read} \in \mathsf{Threats}_A(t)$.

The adversary types define which actions an adversary may apply to a communication link with a given stereotype. delete means that the adversary may delete the messages in the corresponding link queue, read allows him to read the messages in the link queue, and insert allows him to insert messages in the link queue.

To evaluate the security of the system with respect to the given type of adversary, we define the *execution of the subsystem $\mathcal{S}$ in presence of an adversary of type A* to be the function $[\![\mathcal{S}]\!]_A()$ defined from $[\![\mathcal{S}]\!]()$ by applying the adversary function to the link queues as a fourth step in the definition of $[\![\mathcal{S}]\!]()$ as follows:

(4) The type A adversary function is applied to the link queues as detailed above.

The UMLsec profile makes use of a formalization of the security requirement *secrecy* following one of the standard approaches in formal methods: It relies on the idea that a specification preserves the secrecy of some data d if the system never sends out any information from which d could be derived, even in interaction with an adversary (where the *knowledge set* collects the information gained by an adversary).

We say that a subsystem $\mathcal{S}$ *preserves the secrecy* of an expression E from adversaries of type A if E never appears in the knowledge set of A during execution of $[\![\mathcal{S}]\!]_A()$.

3 The UMLsec Extension: Architectural Stereotypes

We shortly recall a simplified fragment of the UMLsec profile that is relevant to secure architectures. A complete account can be found in [5].

For adaption to a particular application domain UML provides three "lightweight" extension mechanisms: *Stereotypes* give a specific meaning to the model elements they are attached to and are represented by double angle brackets. A *tagged value* is a name-value pair in curly brackets associating data with elements in the model. Furthermore, *constraints* may be attached that have to be satisfied by the diagram.

We explain some of the UMLsec stereotypes and tags and give examples. The constraints are parameterized over the adversary type with respect to which the security requirements should hold; we thus fix an adversary type A to be used in the following. Some of the constraints refer to the formal definitions in Sect. 2. They can be checked automatically using the tool-support presented in [5].

Internet, encrypted, LAN

These stereotypes on links in deployment diagrams denote the respective kinds of communication links. We require that each link carries at most one of these stereotypes. For each adversary type A, we have a function $\mathsf{Threats}_A(s)$ from each stereotype $s \in \{\text{《}\mathsf{encrypted}\text{》}, \text{《}\mathsf{LAN}\text{》}, \text{《}\mathsf{Internet}\text{》}\}$ to a set of strings $\mathsf{Threats}_A(s) \subseteq \{\mathsf{delete}, \mathsf{read}, \mathsf{insert}\}$. This way we can evaluate UML specifications using the approach explained in Sect. 2. We make use of this for the constraints of the remaining stereotypes of the profile.

As an example for a threat function, Fig. 1 gives the one for the *default* type of attacker, which represents an outsider adversary with modest capability.

secure links

This stereotype, which may label subsystems, is used to ensure that security requirements on the communication are met by the physical layer. More precisely, the constraint enforces that for each dependency d stereotyped 《secrecy》 between subsystems or objects on different nodes n, m, we have a communication link l between n and m with stereotype s such that $\mathsf{read} \notin \mathsf{Threats}_A(s)$.

Example In Fig. 2, given the *default* adversary type, the constraint for the stereotype 《secure links》 is violated: The model does not provide communication secrecy against the *default* adversary, because the Internet communication link between web-server and client does not provide the needed security level according to the $\mathsf{Threats}_{default}(Internet)$ scenario.

Stereotype	Threats$_{default}$()
Internet	{delete, read, insert}
encrypted	{delete}
LAN	$\emptyset$
wire	$\emptyset$
smart card	$\emptyset$
POS device	$\emptyset$
issuer node	$\emptyset$

Fig. 1. Threats from the *default* attacker

secrecy

«call» or «send» dependencies in object or component diagrams stereotyped «secrecy» are supposed to provide secrecy for the data that is sent along them as arguments or return values of operations or signals. This stereotype is used in the constraint for the stereotype «secure links».

secure dependency

This stereotype, used to label subsystems containing object diagrams or static structure diagrams, ensures that the «call» and «send» dependencies between objects or subsystems respect the security requirements on the data that may be communicated along them. More exactly, the constraint enforced by this stereotype is that if there is a «call» or «send» dependency from an object (or subsystem) C to an object (or subsystem) D then the following conditions are fulfilled.

- For any message name n offered by D, n appears in the tag {secret} in C if

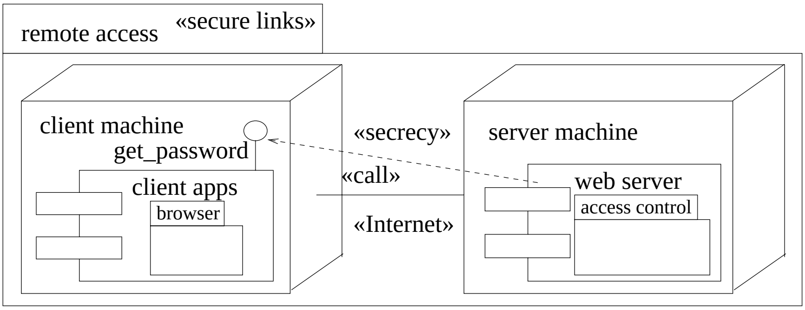

Fig. 2. Example *secure links* usage

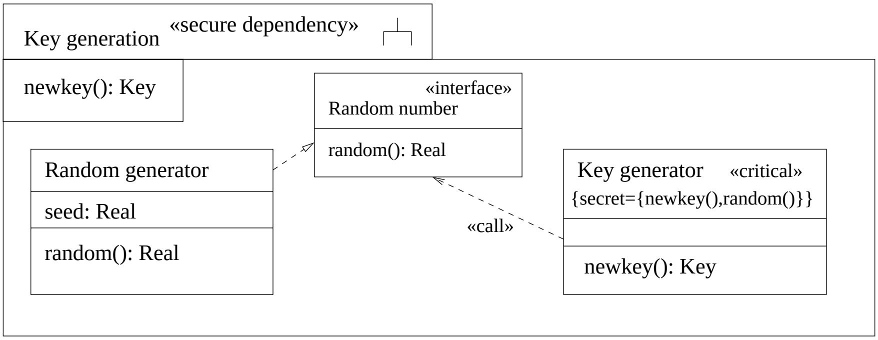

Fig. 3. Key generation subsystem

and only if it does so in D.

- If a message name offered by D appears in the tag {secret} in C then the dependency is stereotyped «secrecy».

Example Figure 3 shows a key generation subsystem stereotyped with the requirement «secure dependency». The given specification violates the constraint for this stereotype, since Random generator and the «call» dependency do not provide the security levels for random() required by Key generator.

critical

This stereotype labels objects whose instances are critical in some way, as specified by the associated tag {secret}, the values of which are data values or attributes of the current object the secrecy of which are supposed to be protected. This protection is enforced by the constraints of the stereotypes «data security» and «no down − flow» (depending on the degree of secrecy required) which label subsystems that contain «critical» objects.

data security

This stereotype labeling subsystems has the following constraint. The subsystem behavior respects the data security requirements given by the stereotype «critical» and the associated tags, with respect to the threat scenario arising from the deployment diagram. More precisely, the constraint is that the stereotyped subsystem preserves the secrecy of the data designated by the tag {secret} against adversaries of type A as defined in Sect. 2.

Example The example in Fig. 4 shows the specification of a simple security protocol. The sender requests the public key K together with the certificate
$\mathcal{S}ign_{K_{CA}}(rcv :: K)$ certifying authenticity of the key from the receiver and sends the data d back encrypted under K (here $\{M\}_K$ is the encryption of

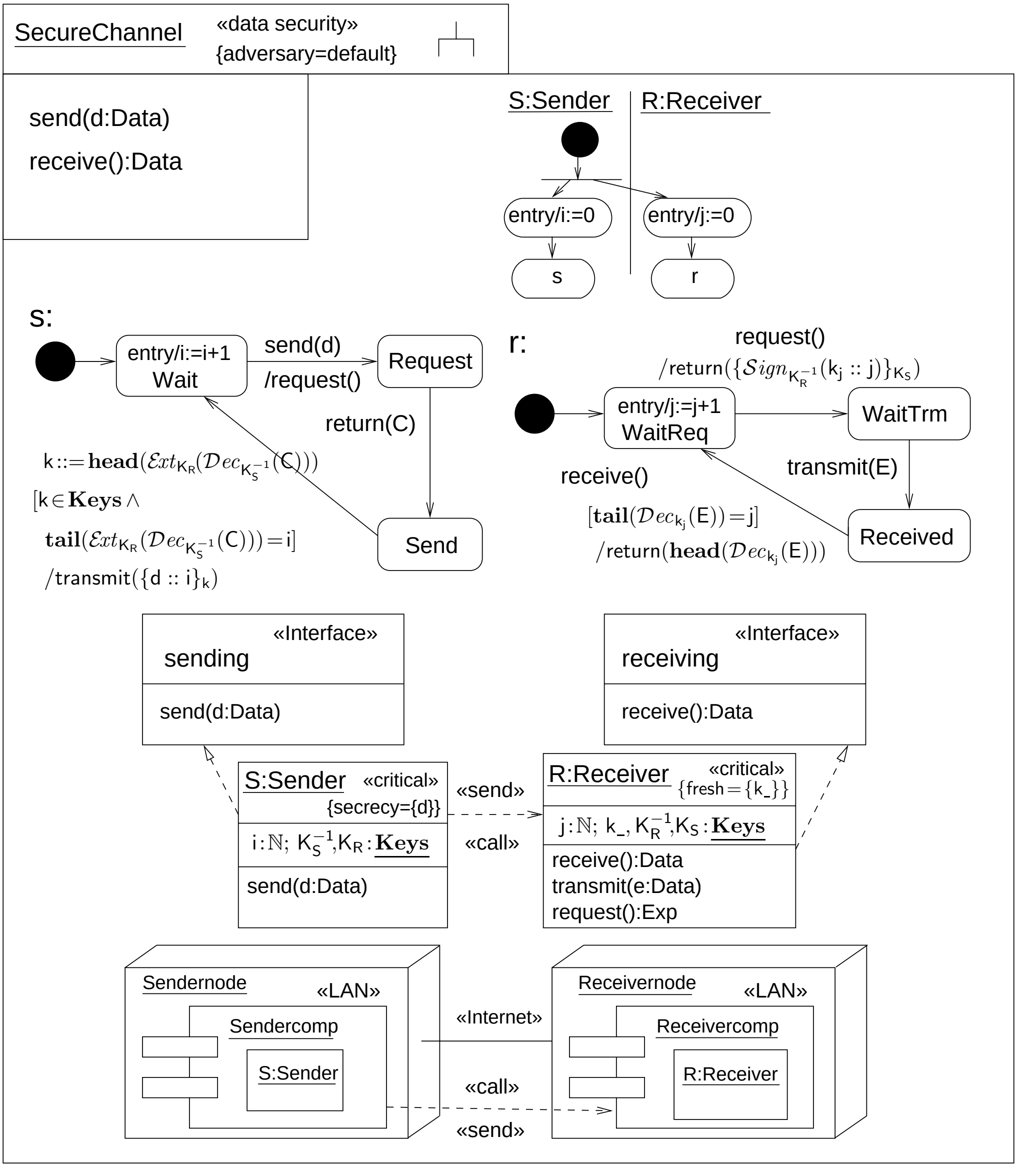

Fig. 4. Security protocol

the message M with the key K, $\mathcal{D}ec_K(C)$ is the decryption of the ciphertext C using K, $\mathcal{S}ign_K(M)$ is the signature of the message M with K, and $\mathcal{E}xt_K(S)$ is the extraction of the data from the signature using K). Assuming the *default* adversary type and by referring to the adversary model outlined in Sect. 2, one can establish that the secrecy of d is preserved.

4 Foundations for Secure Architectural Design Patterns

There are several conceptual aids for designing secure architectures using UMLsec. For example, in [5], we explain how to use tool supported techniques such as refinement and modularity. In this section, we shortly sketch how one could use security patterns in the context of UMLsec.

Software Architecture Patterns [1] encapsulate the design knowledge of software architects by presenting recurring design problems and standardized solutions. One can use transformations of UMLsec models to introduce patterns within the design process. A goal of this approach is to ensure that the patterns are introduced in a way that has previously been shown to be useful and correct. Also, having a sound way of introducing patterns using transformations can ease security analysis, since the analysis can be performed on the more abstract and simpler level, and one can derive security properties of the more concrete level, provided that the transformation has been shown to preserve the relevant security properties.

In our approach, the application of a pattern p corresponds to a function f_p which takes a UML specification $\mathcal{S}$ and returns a UML specification, namely the one obtained when applying p to $\mathcal{S}$. Technically, such a function can be presented by defining how it should act on certain subsystem instances, and by extending it to all possible UML specifications in a compositional way. Suppose that we have a set S of subsystem instances such that none of the subsystem instances in S is contained in any other subsystem instance in S. Suppose that for every subsystem instance $\mathcal{S} \in S$ we are given a subsystem instance $f_p(\mathcal{S})$. Then for any UML specification $\mathcal{U}$, we can define $f_p(\mathcal{U})$ by substituting each occurrence of a subsystem instance $\mathcal{S} \in S$ in $\mathcal{U}$ by $f_p(\mathcal{S})$.The challenge then is to define such a function f_p that is applicable as widely as possible. How to do this on a technical level is beyond the scope of this first introduction to UMLsec. Here we just demonstrate the idea by an example.

Consider the problem of communication over untrusted networks, as exemplified in Fig. 5. A well-known solution to this problem is to encrypt the traffic over the untrusted link using a key exchange protocol, as demonstrated in Fig. 4. A detailed explanation of this pattern is given in [5]. The Secure Channel Pattern could thus be formulated intuitively as follows: In a situation such as the one in Fig. 5, one can implement the secure channel needed to enforce the security requirements using the system in Fig. 4.

To apply this pattern p in a formal way, we consider the set S of subsystems derived from the subsystem in Fig. 5 by renaming: This means, we substitute any message, data, state, subsystem instance, node, or component name n by a name m at each occurrence, in a way such that name clashes are avoided.

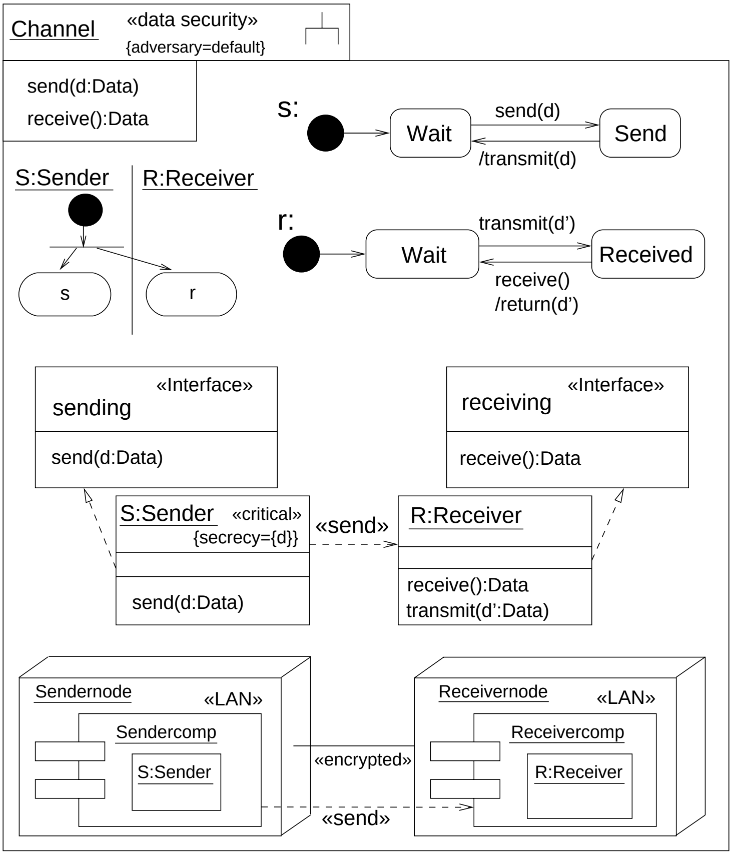

Fig. 5. Secure architecture pattern example: sender and receiver

Then f_p maps any subsystem instance $\mathcal{S} \in S$ to the subsystem instance derived from that given in Fig. 4 by the same renaming. This gives us a presentation of f_p from which the definition of f_p on any UML specification can be derived as indicated above. Since one can show that the subsystem in Fig. 4 is secure in a precise sense, as explained in [5]. this gives one a convenient way of reusing security engineering knowledge in a well-defined way within the development context.

5 Secure Foundations for the Java Security Architecture

In this section, we explain how to provide foundations for designing secure systems based on the Java Security Architecture.

Dynamic access control mechanisms which are part of security architectures such as the JDK 1.2 Security Architecture with its GuardedObjects can be difficult to administer since it is easy to forget an access check [3]. If the appropriate access controls are not performed, the security of the entire system may be compromised. Additionally, access control may be granted indirectly and unintentionally by granting access to an object containing the signature key that enables access to another object. We show how to use UMLsec in the context of the Java Security Architecture to address these problems by providing means of reasoning about the correct deployment of security architectural mechanisms such as guarded objects.

Authorization or *access control* is one of the cornerstones of computer security. The objective is to determine whether the source of a request is *authorized* to be granted the request. Distributed systems offer additional challenges. The trusted computing bases (TCBs) may be in various locations and under different controls. Communication is in the presence of possible adversaries. Mobile code is employed that is possibly malicious. Further complications arise from the need for delegation, meaning that entities may act on behalf of other entities. Also, many security requirements are location-dependent. For example, a user may have more rights at the office terminal than when logging on from home.

In Java, this problem is addressed at the architectural level by providing a Java Security Architecture. In particular, from JDK 1.2, a fine-grained security architecture is employed which offers a user-definable access control, and the sophisticated concept of guarded objects [3]. Permissions are granted to protection domains. A *protection domain* [7] is a set of entities accessible by a principal. In the JDK 1.2, protection domains consist of classes and objects. They are specified depending on the origin of the code, as given by a URL, and on the key with which the code may be signed. The system security policy set by the user or a system administrator is represented by a policy object instantiated from the class java.security.Policy. The security policy maps protection domains to sets of access permissions given to the code. There is a hierarchy of typed and parameterized access permissions, of which the root class is java.security.Permission and other permissions are subclassed either from the root class or one of its subclasses. Permissions consist of a target and an action. For file access permissions in the class FilePermission, the targets can be directories or files, and the actions include read, write, execute, and delete. An access permission is granted if all callers in the current thread history belong to domains that have been granted the said permission. The history of a thread includes all classes on the current stack and also transitively inherits all classes in its parent thread when the current thread is created. If the supplier of a resource is not in the same thread as the consumer, and the consumer thread cannot provide the access control context information, one can use a GuardedObject to protect access to the resource. The supplier of the resource creates an object representing the resource and a GuardedObject containing the resource object, and then hands the GuardedObject to the consumer. A specified Guard object incorporates checks that need to be met so that the resource object can be obtained. For this, the Guard interface contains the method checkGuard, taking an Object argument and performing the checks. To grant access the Guard objects simply returns, to deny access it throws a SecurityException. GuardedObjects are a quite powerful access control mechanism. However, their use can be difficult to administer. For example, guard objects may check the signature on a class file. This way, access to an object may be granted indirectly, and possibly unintentionally, by giving access to another object containing the signature key for which the corresponding signature provides access to the first object.

Thus the sophisticated access control mechanisms of the JDK 1.2 Security Architecture are not so easy to use. The granting of permissions depends on the execution context. Sometimes, access control decisions rely on multiple threads. A thread may involve several protection domains. It is not always easy to see if a given class will be granted a certain permission. In the remainder of this section, we explain some UMLsec stereotypes that support secure use of the Java Security Architecture mechanisms.

guarded access

This stereotype of subsystems is supposed to mean that each object in the subsystem that is stereotyped «guarded» can only be accessed through the objects specified by the tag {guard} attached to the «guarded» object. Formally, we assume that we have $name \notin \mathcal{K}_A^p$ for the adversary type A under consideration and each name *name* of an instance of a «guarded» object, meaning that a reference is not publicly available. Also, we assume that for each «guarded» object there is a statechart specification of an object whose name is given in the associated tag {guard}. This way, we model the passing of references.

We illustrate this stereotype with the example of a web-based financial application. Two institutions offer services over the Internet to local users: an Internet bank, Bankeasy, and a financial advisor, Finance. To make use of these services, a local client needs to grant the applets from the respective sites certain privileges. Access to the local financial data is realized using GuardedObjects. The specification of the local system part is given in Fig. 6. It contains the simplified relevant part of the Java Security Architecture which receives requests for object references and forwards them to the guard objects of the three guarded objects. Since the «guarded» objects StoFi, FinEx, and

MicSi can only be accessed through their associated guard, the subsystem instance fulfills the condition associated with the stereotype «guarded access» with regard to default adversaries. The access controls are realized by the Guard objects FinGd, ExpGd, and MicGd, whose behavior is specified. For example, applets that are signed by the bank can read and write the financial data stored in the local database, but only between 1 pm and 2 pm. This which is enforced by the FinGd guard object, where we assume that the condition slot is fulfilled if and only if the time is between 1 pm and 2 pm.

guarded

This stereotype labels objects in the scope of the stereotype «guarded access» above that are supposed to be guarded. It has a tagged value {guard} which defines the name of the corresponding guard object. As an example, in Fig. 6, the «guarded» objects StoFi, FinEx, and MicSi are protected by the {guard} objects Guard objects FinGd, ExpGd, and MicGd, respectively.

6 Tool Support

To facilitate the application of our approach in industry, automated tools for the analysis of UML models using the suggested semantics are required. We describe a framework that incorporates several such verifiers currently developed at the TU München. More information can be found in [5].

The Fig. 7 illustrates the architecture of the UML tool framework which meets the listed requirements. We briefly describe its functionality. The developer creates a model and stores it in the UML 1.5 / XMI 1.2 file format. The file is imported by the Java-based tool into the MDR repository which is part of the Netbeans library. The tool accesses the model through the JMI interfaces generated by the MDR library. The checker parses the model and checks the constraints associated with the stereotype, by calling sophisticated analysis engines such as the first-order logic automated theorem prover e-Setheo, the model-checker Spin, and Prolog-based analysis routines. The results are delivered as a text report for the developer describing found problems, and a modified UML model, where the stereotypes whose constraints are violated are highlighted.

7 Experience and Outlook

The method proposed here has been successfully applied in secure systems projects, for example in an evaluation of the Common Electronic Purse Specifications under development by Visa International and others, in a project

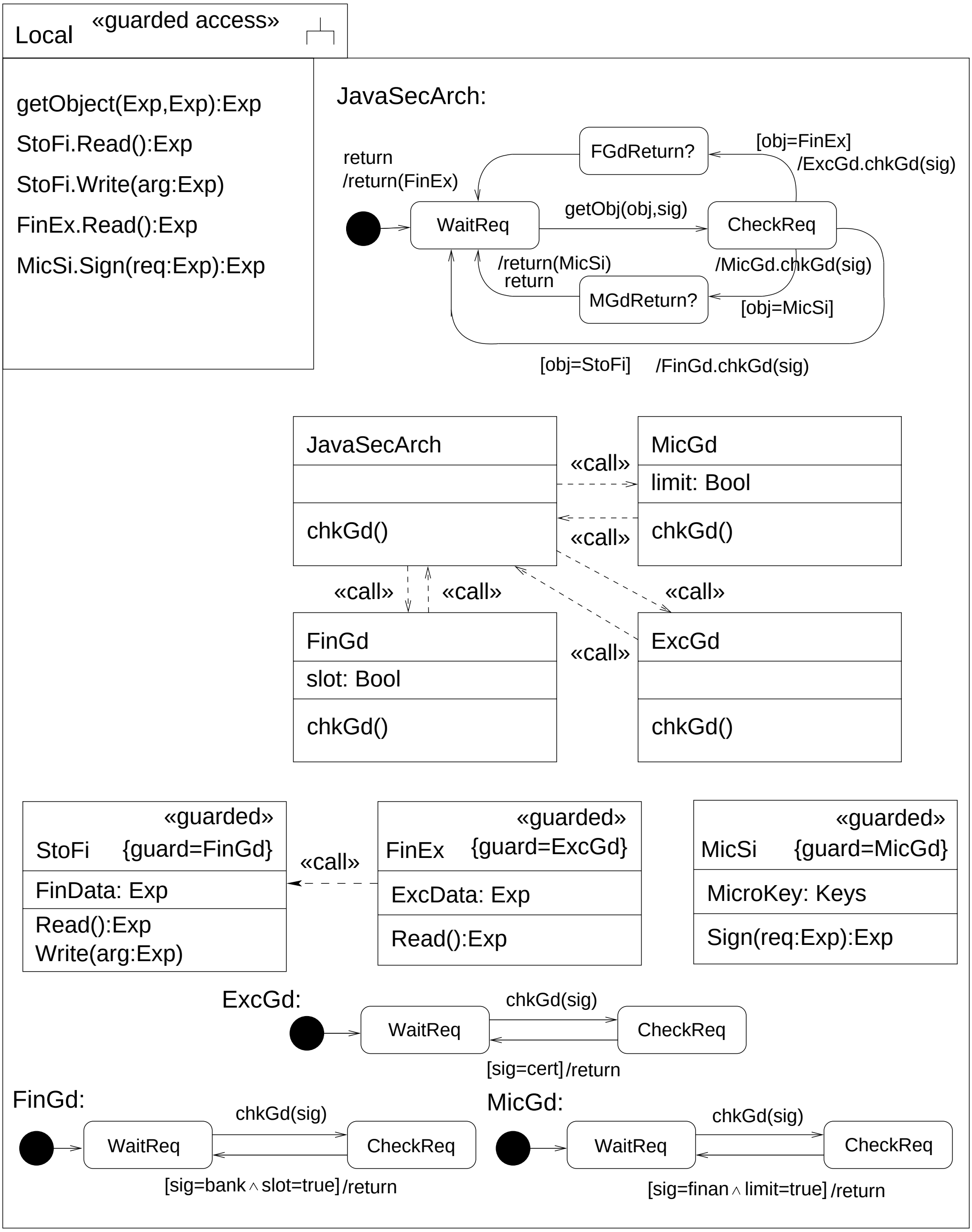

Fig. 6. Financial application specification: Local system

with a large German bank analyzing a security-critical Internet bank architecture, and in projects analyzing a Biometric access control architecture of a German telecom company and an automotive emergeny application of a German car manufacturer within the Verisoft project funded by the German

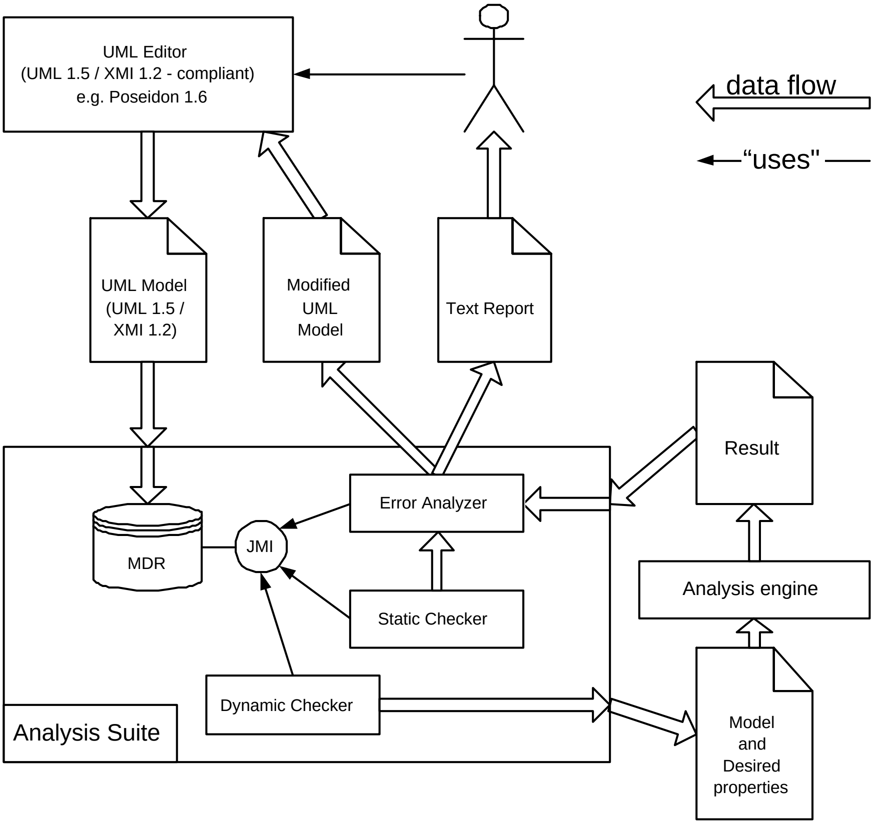

Fig. 7. UML tools suite

Ministry of Science and Technology. In particular, these experiences indicate that the approach is adequate for use in practice.

Given the current state of security-critical systems in practice, with many weaknesses reported continually, it seems to be a promising idea to apply model-driven development to secure systems architecture design, since it enables developers with little background in security to make use of security engineering knowledge encapsulated in a widely used design notation. Since there are many highly subtle security requirements which can hardly be verified with the "bare eye", even critical systems experts may profit from this approach.

For these ideas to be of benefit in practice, it is important to have intelligent tool-support to assist in using them. As sketched above, tools exist that one can use to check the constraints illustrated above mechanically, which supports the approach by saving time and preventing errors when analyzing the model for security design flaws.

More information can be found in the book [5] and articles including [4].

References

[1] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[2] Garlan, D., Software architecture: a roadmap. In *22nd International Conference on on Software Engineering (ICSE 2000): Future of Software Engineering Track*, pages 91–101. ACM, 2000.

[3] Gong, L., *Inside Java 2 Platform Security – Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA, 1999.

[4] Jürjens, J., UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425. Springer-Verlag, 2002.

[5] Jürjens, J., *Secure Systems Development with UML*. Springer-Verlag, 2004.

[6] Jürjens, J., Software Architectures for Safe and Secure Systems. In *5th IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE Computer Society, 2004. Half-day tutorial.

[7] Saltzer, J., and M. Schroeder, The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.